---

title: Confidential Compute Requests description: How to understand and make confidential compute requests on SUAVE keywords: - practical - suave - confidential - example

---

:::info

There are a number of different ways to work with SUAVE and to craft Confidential Compute Requests (CCRs). We'll discuss how to use both the TypeScript and Golang SDKs in this tutorial.

:::

TypeScript
For web developers

Golang
For engineers

TypeScript

The two templates from the previous tutorial illustrate different ways of using CCRs on SUAVE.

The **Next** template uses a basic contract to demonstrate the absolute basics of how CCRs work and how to use them as a developer building SUAPPs.

The **TypeScript** template will lead you through how to sign a transaction on another domain (Goerli in this case), and then submit that signed transaction in the data of your CCR on SUAVE.

Callbacks

Let's start in the Next template and look at how CCRs work and how to use them.

OnChainState.sol demonstrates that any CCR which tries to change state directly will revert. Rather, you need to use a callback to a different function that does change state in order to ensure that data sent in a CCR remains confidential:

```solidity // nilExample is a function executed in a confidential request // that CANNOT modify the state of the smart contract. function nilExample() external payable returns (bytes memory) { require(Suave.isConfidential()); state++; return abi.encodeWithSelector(this.nilExampleCallback.selector); }

function exampleCallback() external {
    state++;
    emit UpdatedState(state);
}

// example is a function executed in a confidential request that includes
// a callback that can modify the state.
function example() external view returns (bytes memory) {
    require(Suave.isConfidential());
    return bytes.concat(this.exampleCallback.selector);
}

```

If you try and call nilExample() from the frontend, it will revert. And, in order to call example(), we need to understand how to craft a CCR so that we can pass the require(Suave.isConfidential()); check. The code required for this is here:

ts const sendExample = async () => { if (!provider || !suaveWallet) { console.warn(`provider=${provider}\nsuaveWallet=${suaveWallet}`) return } const nonce = await provider.getTransactionCount({ address: suaveWallet.account.address }); const ccr: TransactionRequestSuave = { confidentialInputs: '0x', kettleAddress: '0xB5fEAfbDD752ad52Afb7e1bD2E40432A485bBB7F', // Use 0x03493869959C866713C33669cA118E774A30A0E5 on Rigil. to: deployedAddress, gasPrice: 2000000000n, gas: 100000n, type: '0x43', // SuaveTxRequestTypes.ConfidentialRequest chainId: 16813125, // chain id of local SUAVE devnet and Rigil data: encodeFunctionData({ abi: OnChainState.abi, functionName: 'example', }), nonce }; const hash = await suaveWallet.sendTransaction(ccr); console.log(`Transaction hash: ${hash}`); setPendingReceipt(provider.waitForTransactionReceipt({ hash })); }

There are a few key points to understand:

1. You need to enable "Eth_sign" in the Advanced settings in your browser wallet for this to work.
2. We can leave the confidentialInputs field empty, as we're not actually sending any data along with this transaction: just demonstrating how CCRs work.
3. We specify a new type for the transaction. The different suave-viem transaction types are defined here.
4. The suaveWallet.sendTransaction(ccr) uses eth_sign under the hood, along with a few other steps required to serialize the

transaction correctly, which you can see happening here.

CCRs with data

What happens when we do actually want to send data in our CCR?

The TypeScript template demonstrates how to do this by signing transactions on Goerli that you wish to be processed by Kettles on SUAVE.

In this case, we need to:

1. Craft a transaction on Goerli and sign it.
2. Append relevant details like the `decryptionCondition`, `kettleAddress`, `contract` and `chainId`.
3. Use a helper function like `toConfidentialRequest` to
    1. place our signed transaction in `confidentialInputs` and
    2. replace the `data` field with a call to the appropriate method in the specified SUAVE contract.

The code which achieves this can be found here and looks like this:

```ts const sendDataRecord = async (suaveWallet: any) => { // create sample transaction; won't land onchain, but will pass payload validation const sampleTx = { type: "eip1559" as 'eip1559', chainId: 5, nonce: 0, maxBaseFeePerGas: 0x3b9aca00n, maxPriorityFeePerGas: 0x5208n, to: '0x0000000000000000000000000000000000000000' as Address, value: 0n, data: '0xf00ba7' as Hex, } const signedTx = await goerliWallet.signTransaction(sampleTx) console.log("signed goerli tx", signedTx)

// create dataRecord & send ccr
try {
    const dataRecord = new MevShareRecord(
        1n + await goerliProvider.getBlockNumber(),
        signedTx,
        KETTLE_ADDRESS,
        MevShareContract.address as Address,
        suaveRigil.id
    )
    console.log(dataRecord)
    const ccr = dataRecord.toConfidentialRequest()
    const txHash = await suaveWallet.sendTransaction(ccr)
    console.log("sendResult", txHash)
    // callback with result
    onSendDataRecord(txHash)
} catch (e) {
    return onSendDataRecord('0x', e)
}

} ```
```

Here, we use the convenient `toConfidentialRequest()` request method once we have the signed transaction, rather than crafting it manually ourselves and setting the transaction type as we did in the Next template.

This should give you enough information to start crafting your own CCRs and learning how to leverage the benefits of credible, confidential computation.

Golang

:::info

The **SUAPP Examples repo** uses the Golang SDK extensively and is a good reference for best practices.

:::

Let's look at the private order-flow auction example to understand how to use the Golang SDK to craft more complex kinds of CCRs.

The flow of this example is similar to the TypeScript MEVShare demo directly above. However, we'll use it to demonstrate a full e2e flow with CCRs:

1. Deploy the OFA contract.
2. Instantiate two new accounts and fund them.
3. Craft two different transactions: `ethTxn1` which simulates a transaction from another domain, and `ethTxnBackrun` which simulates a backrun based on the hint emitted by `ethTxn1`.
4. Wrap the first transaction into a "bundle", add any extra data required (in this case, just the percentage of MEV the user is willing to share with whomever includes their transaction), and send it to the OFA contract on SUAVE.
    1. Receive and store the hint event that is emitted as a result of the transaction.

5. Wrap the second transaction into another bundle and send it to the OFA contract (along with extra data like the hint event ID).
    1. Receive and store the match event in the case that the contract does batch the transactions together.
6. Send that resulting batch of matched transactions to a known relayer off-chain so that it can be included on the intended domain.

In order to create a signed transaction on Ethereum L1, you can use the Golang SDK like this:

```
go ethTxn1, _ := fr.L1.SignTx(testAddr1, &types.LegacyTx{ To: &targeAddr, Value: big.NewInt(1000), Gas: 21000, GasPrice: big.NewInt(670189871), })
```

In order to create a bundle which includes this transaction (and any others which may be relevant to you), you define it as a type and include the transactions you're interested in, along with any other relevant data:

```
go bundle := &types.SBundle{ Txs: types.Transactions{ethTxn1}, RevertingHashes: []common.Hash{}, RefundPercent: &refundPercent, } bundleBytes, _
:= json.Marshal(bundle)
```

In order to send this bundle to the contract on SUAVE, passing in particular arguments and storing the resulting receipt, we do this:

```
go receipt = contract.SendTransaction("newMatch", []interface{}{hintEvent.DataRecordId}, backRunBundleBytes)
```

We can visualize the most general form of this example (involving multiple different transactions types, actors, and actions) like this: